Ответ на вопрос по выбору по теме "Паттерн Object Pool"

Выполнил:
студент 2 курса,
гр. БПМ-22-4 Воеводин Е.О.
29.12.2023

Паттерн Object Pool

Определение

Является порождающим паттерном, используемым для получения готового количества к использованию объектов. Вместо уничтожения объекта - этот объект возвращается обратно в пул.
Пул стоит использовать тогда, когда объектов очень много, и они создаются и удаляются постоянно. При этом как пишут даже в Unity: "...потому что пулы не должны быть вашим дежурным решением."

Примеры

- Doom. Бесспорно самый популярный пример. Там, как и во многих играх того поколения, пул использовался для хранения видимых объектов.
- Большие карты в играх также могут использовать пулинг для хранения информации о всех объектах.
- Object Pool был использован в моей курсовой для создания постоянного потока пуль(200+ пуль, которые постоянно создаются и удаляются)

Интерфейс

```
template<typename T>
class IObjectPool  {
public:
        // используем std::function<void(T*)>, т.к. нам нужна функция, которая
будет при удалении возвращать объект в пул.
        using Ptr = std::unique_ptr<T, std::function<void(T*)>>;
        // Дефолтный конструктор
        IObjectPool() = default;
        // Обозначен virtual, т.к. наследование.
        virtual ~IObjectPool() = default;

        // Мы не хотим разрешать пользователю копировать пул.
        IObjectPool(const IObjectPool&) = delete;
        IObjectPool& operator=(const IObjectPool&) = delete;
```

```
        // Moving is ok!
        IObjectPool(IObjectPool&&) = default;
        IObjectPool& operator=(IObjectPool&&) = default;

        // Добавляем объект в пул, при этом забирая владение этим
        // объектом себе.
        void pushObject(std::unique_ptr<T> obj);
        // Получаем последний добавленный объект, не передавая владение
пользователю
        Ptr [[nodiscard]] extractObject();
        // Размер пула.
        std::size_t [[nodiscard]] size() const noexcept;
        // Проверка на пустоту пула.
        bool [[nodiscard]] isEmpty() const noexcept;
        // Полностью очищаем пул.
        void dispose();
};
```

Реализация

Самая простая реализация, которую мы назовём NaiveObjectPool, работает достаточно легко:

```
template<typename T>
class NaiveObjectPool final {
public:
        //...code before

        // Просто используем функции контейнера(в даном случае стека)
        size_t [[nodiscard]] size() const noexcept { return objects_.size(); }
        bool [[nodiscard]] isEmpty() const noexcept { return objects_.empty(); }
        //...
private:
        std::stack<std::unique_ptr<T>> objects_;
};
```

```
template<typename T>
[[nodiscard]] std::unique_ptr<T> NaiveObjectPool<T>::extractObject()
{
        // Т.к. нельзя получить к объект доступ, если пул пуст - ждём новый.
        if (objects_.empty()) {
                throw std::exception("Tried to call extraction on an empty pool!");
        }
        // Забираем объект, при этом добавляя ему функцию, которая будет
```

```
        // возвращать его в пул по убийству.
        Ptr object(objects_.top().release(),
                [this](T* ptr) {
                this->add(std::unique_ptr<T>);
                });
        // Удаляем его из нашего стека
        objects_.pop();
        return object;

}

template<typename T>
void NaiveObjectPool<T>::dispose()
{
        for (; !objects_.empty(); objects_.pop());
}

template<typename T>
void NaiveObjectPool<T>::pushObject(std::unique_ptr<T> obj)
{
        objects_.push(std::move(obj));
}
```

Пример использования:

```
NaiveObjectPool<int> pool;
CHECK(pool.isEmpty() == true);
CHECK(pool.size() == 0);
pool.pushObject(std::make_unique<int>(42));
pool.pushObject(std::make_unique<int>(228));
CHECK(pool.isEmpty() == false);
CHECK(pool.size() == 2);
auto object_228 = pool.extractObject();
CHECK(pool.isEmpty() == false);
CHECK(pool.size() == 1);
auto object_42 = pool.extractObject();
CHECK(pool.isEmpty() == true);
CHECK(pool.size() == 0);
CHECK(*object_42 == 42);
CHECK(*object_228 == 228);
```

Однако у такого подхода есть проблема:
Если создавать pool через указатель, то может случиться такое, что пул удалится до того, как все объекты успеют вернуться туда, тем самым создавая memory leak.

Более защищенный вариант

```
template<typename T>
class ObjectPool final {
public:
        // Создаем функтор Делетер, который будет за нас проверять, жив ли пул или
нет. И возможно ли обратно запушить объект
        class Deleter {
        public:
                explicit Deleter(std::weak_ptr<ObjectPool*> pool) : pool_(pool) {}
                void operator()(auto* ptr) {
                        if (auto locked_pool = pool_.lock()) {
                                (*locked_pool.get())->pushObject(std::unique_ptr<T>
(ptr));
                        }
                        else {
                                std::default_delete<T> default_deleter;
                                default_deleter(ptr);
                        }
                }
        private:
                // weak_ptr, т.к. пула может уже и не существовать.
                std::weak_ptr<ObjectPool*> pool_;
        };
        using Ptr = std::unique_ptr<T, Deleter>;


        ObjectPool() : this_shared_(new ObjectPool*(this)) {}
        virtual ~ObjectPool() = default;

        ObjectPool(const ObjectPool&) = default;
        ObjectPool(ObjectPool&&) = default;


        void pushObject(std::unique_ptr<T> obj) { objects_.push(std::move(obj)); }
;
        Ptr [[nodiscard]] extractObject() {

                if (objects_.empty()) {
                        throw std::exception("Tried to call extraction on an empty
pool!");
                }

                Ptr top(objects_.top().release(),
```

```
Deleter{std::weak_ptr<ObjectPool*>{this_shared_}});
                objects_.pop();
                return std::move(top);
        }
        size_t [[nodiscard]] size() const noexcept { return objects_.size(); };
        bool [[nodiscard]] isEmpty() const noexcept { return objects_.empty(); };

        void dispose();
private:
        std::stack<std::unique_ptr<T>> objects_;
        std::shared_ptr<ObjectPool*> this_shared_;
};
```

Но есть ещё!

- Хороший вариант, который не успел рассмотреть в данном вопросе: Singletone Object Pool. Следуя из названия -- это просто пул, написанный как синглтон(другой порождающий паттерн)

Источники:

- Перевод видео юнити на хабре
- Статья на сайте юнити
- Хороший пример написания пула